StockFlow Case Study — Submission

Candidate: Mujjamil Sofi
Role: Backend / B2B SaaS Engineer
Project: StockFlow — Inventory Management Platform

Part 1 — Code Review and Debugging

Problem Statement

A previous intern implemented the `create_product` API endpoint. Although the code compiles, it fails under production conditions and does not fully satisfy business requirements.

Original Implementation:

```
@app.route('/api/products', methods=['POST'])
def create_product():
    data = request.json

    product = Product(
        name=data['name'],
        sku=data['sku'],
        price=data['price'],
        warehouse_id=data['warehouse_id']
    )

    db.session.add(product)
    db.session.commit()

    inventory = Inventory(
        product_id=product.id,
        warehouse_id=data['warehouse_id'],
        quantity=data['initial_quantity']
    )

    db.session.add(inventory)
    db.session.commit()

    return {"message": "Product created", "product_id": product.id}
```

Issues Identified

Technical Issues:

1. No input validation; missing fields cause runtime errors.

2. Two separate commits; failure in inventory creation leaves inconsistent data.
3. SKU uniqueness not enforced; duplicate SKUs possible.
4. Product tied to a single warehouse; requirement specifies multi-warehouse support.
5. Price stored as raw value; floating-point may lead to rounding errors.
6. Optional fields not handled; absence of `initial_quantity` causes KeyError.
7. No transaction rollback; partial failures leave database in inconsistent state.

Business Logic Issues:

1. Multi-warehouse support broken.
2. Partial failures create orphaned products.
3. Duplicate SKUs compromise order processing and accounting.
4. API behavior is non-deterministic under invalid inputs.

Corrected Implementation

Goals:

- Atomic transaction to prevent partial writes.
- Input validation with clear error messages.
- SKU uniqueness enforcement.
- Support for optional inventory initialization.
- Safe handling of missing or optional fields.

```
@app.route('/api/products', methods=['POST'])
def create_product():
    data = request.get_json() or {}

    # Validate required fields
    required = ["name", "sku"]
    missing = [f for f in required if f not in data]

    if missing:
        return {"error": f"Missing fields: {', '.join(missing)}"}, 400

    # SKU uniqueness check
    existing = Product.query.filter_by(sku=data["sku"]).first()
    if existing:
        return {"error": "SKU already exists"}, 409

    try:
        product = Product(
            name=data["name"],
            sku=data["sku"],
            price=data.get("price", 0.00)
        )

        db.session.add(product)
```

```
        if "warehouse_id" in data and "initial_quantity" in data:
            inventory = Inventory(
                product_id=product.id,
                warehouse_id=data["warehouse_id"],
                quantity=data.get("initial_quantity", 0)
            )
            db.session.add(inventory)

        db.session.commit()

        return {
            "message": "Product created",
            "product_id": product.id
        }, 201

    except Exception as e:
        db.session.rollback()
        return {"error": str(e)}, 500
```

Key Fixes Summary:

- Input validation and clear error responses.
- SKU uniqueness enforcement.
- Single transaction for atomic operation.
- Optional inventory creation.
- Rollback on failure.
- Multi-warehouse support preparation.

Assumptions in Part 1:

- SKU uniqueness enforced at the database level.
- Price is stored as a decimal number.
- Product creation allowed without inventory.
- Warehouse must exist before assigning inventory.

Part 2 — Database Design

Requirements Interpreted

- Companies can have multiple warehouses.
- Products can be stored in multiple warehouses.
- Track inventory changes over time.
- Suppliers provide products.
- Some products may be bundles containing other products.

Database Schema

companies

```
CREATE TABLE companies (
    id BIGINT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

warehouses

```
CREATE TABLE warehouses (
    id BIGINT PRIMARY KEY,
    company_id BIGINT NOT NULL,
    name VARCHAR(255) NOT NULL,
    location VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (company_id) REFERENCES companies(id)
);
```

products

```
CREATE TABLE products (
    id BIGINT PRIMARY KEY,
    company_id BIGINT NOT NULL,
    name VARCHAR(255) NOT NULL,
    sku VARCHAR(100) UNIQUE NOT NULL,
    price DECIMAL(12,2),
    product_type VARCHAR(50) DEFAULT 'simple',
    low_stock_threshold INT DEFAULT 10,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (company_id) REFERENCES companies(id)
);
```

inventory

```
CREATE TABLE inventory (
```

```
    id BIGINT PRIMARY KEY,
    product_id BIGINT NOT NULL,
    warehouse_id BIGINT NOT NULL,
    quantity INT NOT NULL DEFAULT 0,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(product_id, warehouse_id),
    FOREIGN KEY (product_id) REFERENCES products(id),
    FOREIGN KEY (warehouse_id) REFERENCES warehouses(id)
);
```

inventory_history

```
CREATE TABLE inventory_history (
    id BIGINT PRIMARY KEY,
    product_id BIGINT NOT NULL,
    warehouse_id BIGINT NOT NULL,
    change_type VARCHAR(50),
    quantity_change INT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (product_id) REFERENCES products(id),
    FOREIGN KEY (warehouse_id) REFERENCES warehouses(id)
);
```

suppliers

```
CREATE TABLE suppliers (
    id BIGINT PRIMARY KEY,
    company_id BIGINT NOT NULL,
    name VARCHAR(255) NOT NULL,
    contact_email VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (company_id) REFERENCES companies(id)
);
```

product_suppliers

```
CREATE TABLE product_suppliers (
    product_id BIGINT NOT NULL,
    supplier_id BIGINT NOT NULL,
    PRIMARY KEY(product_id, supplier_id),
    FOREIGN KEY(product_id) REFERENCES products(id),
    FOREIGN KEY(supplier_id) REFERENCES suppliers(id)
);
```

product_bundles

```
CREATE TABLE product_bundles (
    bundle_id BIGINT NOT NULL,
    component_product_id BIGINT NOT NULL,
    quantity INT NOT NULL,
    PRIMARY KEY(bundle_id, component_product_id),
    FOREIGN KEY(bundle_id) REFERENCES products(id),
    FOREIGN KEY(component_product_id) REFERENCES products(id)
);
```

Design Decisions

- Normalization reduces redundancy.
- Foreign key constraints ensure referential integrity.
- Indexes:
  - `products(sku)`
  - `inventory(product_id, warehouse_id)`
  - `inventory_history(product_id, created_at)`
- Multi-tenant SaaS ready.

Gaps / Questions

1. Can inventory be negative (backorders)?
2. Does price vary per supplier or per warehouse?
3. Are bundles dynamically calculated or pre-stocked?
4. Should thresholds differ per warehouse?
5. Do we need reserved stock tracking?

Assumptions in Part 2:

- Each company has isolated data using `company_id`.
- Product bundles are optional and may not affect individual inventory directly.
- Suppliers can provide multiple products; one product can have multiple suppliers.
- Inventory updates are tracked for audit purposes.

Part 3 — Low-Stock Alerts API

Endpoint:

`GET /api/companies/{company_id}/alerts/low-stock`

Business Rules

- Alert only if stock < threshold.
- Only for products with recent sales activity (last 30 days).
- Supports multiple warehouses per company.
- Includes supplier information.
- Estimates days until stockout.

Assumptions

- Recent sales are stored in a `sales` table with `product_id`, `warehouse_id`, `quantity`, `created_at`.
- Threshold is per product; no per-warehouse override unless specified.
- Bundles are excluded from alerts.
- Days until stockout uses a 30-day average sales calculation.
- If no supplier exists, supplier info is returned as null.

Implementation (Flask)

```
@app.route("/api/companies/<int:company_id>/alerts/low-stock",
methods=["GET"])
def get_low_stock_alerts(company_id):
    thirty_days_ago = datetime.utcnow() - timedelta(days=30)

    results = (
        db.session.query(
            Product.id,
            Product.name,
            Product.sku,
            Product.low_stock_threshold,
            Inventory.quantity,
            Warehouse.id.label("warehouse_id"),
            Warehouse.name.label("warehouse_name"),
        )
        .join(Inventory, Inventory.product_id == Product.id)
        .join(Warehouse, Warehouse.id == Inventory.warehouse_id)
        .filter(Product.company_id == company_id)
        .filter(Inventory.quantity < Product.low_stock_threshold)
        .all()
```

```
    )

    alerts = []

    for row in results:
        sales = (
            db.session.query(func.sum(Sale.quantity))
            .filter(Sale.product_id == row.id)
            .filter(Sale.warehouse_id == row.warehouse_id)
            .filter(Sale.created_at >= thirty_days_ago)
            .scalar()
        )

        if not sales:
            continue

        avg_daily = sales / 30
        days_left = int(row.quantity / avg_daily) if avg_daily else None

        supplier = (
            db.session.query(Supplier)
            .join(ProductSupplier, ProductSupplier.supplier_id ==
Supplier.id)
            .filter(ProductSupplier.product_id == row.id)
            .first()
        )

        alerts.append({
            "product_id": row.id,
            "product_name": row.name,
            "sku": row.sku,
            "warehouse_id": row.warehouse_id,
            "warehouse_name": row.warehouse_name,
            "current_stock": row.quantity,
            "threshold": row.low_stock_threshold,
            "days_until_stockout": days_left,
            "supplier": {
                "id": supplier.id,
                "name": supplier.name,
                "contact_email": supplier.contact_email
            } if supplier else None
        })

    return {
        "alerts": alerts,
        "total_alerts": len(alerts)
    }
```

Edge Cases Handled

- Product below threshold but no recent sales → skipped.
- Supplier missing → null supplier info.
- Division by zero avoided.
- Company isolation enforced to prevent cross-tenant data leakage.

Conclusion

The solution provides:

- Correct and reliable product creation and inventory handling.
- Flexible multi-warehouse inventory tracking.
- Low-stock alerts considering business rules and supplier info.
- Scalable database design with proper constraints and indexes.